Deep Learning

CSCI 1470/2470
Spring 2023

Ritambhara Singh

February 27, 2023
Monday

DALL-E 2 prompt “a painting of deep underwater with a yellow submarine in the bottom right corner”
Recap

Architecture

AlexNet + Pooling

CNNs are “sort of” translationally invariant

Many layers = vanishing gradient

ResNet + Residual blocks

Batch normalization

Deeper CNNs
MNIST & CIFAR are really nice datasets!

- There’s a clear “absolute truth”
  - A 4 is a 4, and a 5 is a 5
- Labels are guaranteed to be good
- Each class is well-represented – there’s lots of data for each class
- Lots of data in general
...most data isn’t so nice
Examples of messy data

IMDB Movie Review Data

In the training set: 1. there is some great builder wisdom in this movie... whereas other biographies of famous people tend to get very poor. This movie always stays focused and gives a good and honest portrayal of the late time. "right things being right. It's a great movie, and I really enjoyed it." (The rating is 10.)
2. this is a great movie! It is so funny! the love story is excellent! I saw this movie when I was a kid and I saw it again the other day, it's just as good. Robert Zemeckis has made a wonder of an 80s movie. It's just superb. watch it! watch it! watch it! highly recommended a long with a movie like "the raiders of the lost ark." (The rating is 10.)
3. I thought this was a very funny movies in the 80s style although it was finished in 1991, perhaps the title is a bit stupid and difficult to remember... and all in all this movie is a great view into a time before the home computer got customary, internet, cell phones and so forth, moreover, the old lady who plays the babysitter is really funny. (The rating is 8.)

In the test set: this is a stunningly beautiful movies the music by philip glass is just a work of pure genius. i can watch this movie again and again... how was it filmed? it's so amazing. if you have not seen this film watch it... again and again! this must be the only movie which in a powerful way... watch it! watch it! watch it! (The rating is 10.)

Astronomy data

DNA sequence data
Dealing with messy data: preprocessing

- Always necessary (unless it’s preprocessed for you…)
- Goal: get messy, unstructured data into \textbf{usable} data for your model
  - Ultimately, data must be converted into \textbf{tensors} to be fed to your model
- What exactly you need to do is dataset- and context- dependent
  - Language models: tokenization, UNKing, ...
  - Quantify non-numerical entities (e.g. categories $\rightarrow$ one-hot vectors)
  - Drop outliers?
  - Normalize/standardize inputs
Dealing with messy data: preprocessing

- Always necessary (unless it’s preprocessed for you…)

- Goal: get messy, unstructured data into **usable** data for your model
  - Ultimately, data must be converted into **tensors** to be fed to your model

- What exactly you need to do is dataset- and context- dependent
  - Language models: tokenization, UNKing, ...
  - Quantify non-numerical entities (e.g. categories → one-hot vectors)
  - Drop outliers? (Does this mean there’s a group your model systematically fails for?)
  - Normalize/standardize inputs

*Does your quantification make sense? What assumptions does it make about the world – e.g. gender binary or racial categories?*
Consequences of complex data...

• Demo
Consequences of complex data...

What is the simplest thing to do here?

This is an example of overfitting
Overfitting: What can we do about it?

1. Early stopping
Early stopping: pseudocode

curr_valid_loss = inf
for i in range(n_epochs):
    train model()
    new_valid_loss = model.get_test_loss()
    if new_valid_loss > curr_valid_loss:
        break
    else:
        curr_valid_loss = new_valid_loss
Early stopping: thoughts

This is kind of a hack...

Can we stop validation loss from increasing altogether?

Any ideas?
Overfitting: What can we do about it?

1. Early stopping
2. Reduce parameters
Reduce parameters - why?

More parameters = more ‘knobs’ to fiddle = more possibilities to learn something overly-specific to the training data

• Example: curve fitting
Reduce parameters - why?

More parameters = more ‘knobs’ to fiddle = more possibilities to learn something overly-specific to the training data

• Example: curve fitting

\[ y = 3x^2 + 2x + 4 \]
Reduce parameters - why?

More parameters = more ‘knobs’ to fiddle = more possibilities to learn something overly-specific to the training data

• Example: curve fitting

\[ y = 3x^2 + 2x + 4 \]  \[ y = 2x^3 - 4x^2 + x - 3 \]
Reduce parameters - why?

More parameters = more ‘knobs’ to fiddle = more possibilities to learn something overly-specific to the training data

• Example: curve fitting

\[ y = 3x^2 + 2x + 4 \]

\[ y = 2x^3 - 4x^2 + x - 3 \]

\[ y = 7x^{10} + x^9 + 6x^8 - 3x^7 - 2x^6 \ldots + x - 3 \]
Reduce parameters in a Neural Net: How?

- Fully connected nets?
- CNN?
- *Any* NN?
Reduce parameters in a Neural Net: How?

- Fully connected:

What can be done here?
Reduce parameters in a Neural Net: How?

- Fully connected: reduce layer size
Reduce parameters in a Neural Net: How?

- CNN:

What about here?
Reduce parameters in a Neural Net: How?

- CNN: decrease num_channels (and also possibly filter_size)
Reduce parameters in a Neural Net: How?

- All architectures: Decrease number of layers
Reduce parameters in a Neural Net: How?

- All architectures: Decrease number of layers

![Diagram of neural network with input, linear layer, and softmax output]
Reduce parameters in a Neural Net: How?

- All architectures: Decrease number of layers
Reduce parameters in a Neural Net: How?

- All architectures: Decrease number of layers
Reducing parameters: Why not?

- Seen (by deep learning folks) as a bit ‘old-fashioned’
  - Classical perspective: model complexity should match data complexity
  - Deep learning perspective: all real-world data is infinitely complex (i.e. there are infinite variations on what a handwritten digit can look like). If your model is overfitting, that just means you don’t have enough data—so get more!

- What if we can’t get more data?
  - (e.g. it’s prohibitively expensive to gather more data)

*Synthesize* variations on your data
Overfitting: What can we do about it?

1. Early stopping
2. Reduce parameters
3. Data augmentation
Data Augmentation

• Generate (random) variations on your training data, treat that as more training data
  • Assumption: your (random) variations still produce data that is within the expected distribution of training data (so you can’t get too crazy)

• Most commonly used on image data
Data Augmentation: Image Examples

Geometric Transformations

- Original
- Horizontal Flip
- Pad & Crop
- Rotate

Filters & Pixel Intensity Transforms
- Blur, sharpen, contrast adjust, ...

Fancy (Learned) Semantic Transforms
(i.e. “Image synthesis for data augmentation”)

- Winter Yosemite → Summer Yosemite
- Summer Yosemite → Winter Yosemite

https://junyanz.github.io/CycleGAN/

https://bair.berkeley.edu/blog/2019/06/07/data_aug/

https://towardsdatascience.com/data-augmentation-for-deep-learning-4fe21d1a4eb9
Data Augmentation: Limitations

• Your model might still overfit!
  • In general, it’s impossible to design an augmentation procedure that covers all the dimensions of variation your data might experience
  • Your model can still overfit to patterns in the dimensions that you don’t augment with variations...
Overfitting: What can we do about it?

1. Early stopping
2. Reduce parameters
3. Data augmentation
4. Dropout
Dropout – general intuition

- Preventing the network from learning under perfect conditions; that is, make it **harder** for the network to learn

A climbing analogy:

**A person is climbing a wall using holds**

- What if, I make a rule that she can climb
- ... only using certain holds (say just green ones!)
- If she can learn to do this using fewer holds...
- ...she’ll definitely be able to do it with ALL the holds
- (learn better climbing techniques in the process)

Dropout $\sim=$ using only a certain holds instead of ALL the holds
Dropout - what?

Typical NN: the output of every node in every layer is used in the next layer of the network.
Dropout: *in a single training pass*, the output of randomly selected nodes from each layer will “drop out”, i.e. be set to 0.
Dropout - what?

Dropout: *in a single training pass*, the output of randomly selected nodes from each layer will “drop out”, i.e. be set to 0.
Dropout - what?

Not just limited to the input layer: can do this to *any* layer of the network
Dropout - what?

The nodes that drop out will be different each pass (re-randomly selected)
Dropout - what?

The nodes that drop out will be different each pass (re-randomly selected)
Dropout - why?

- Sort of looks like data augmentation, if you squint hard enough
  - Augmenting the data by randomly dropping out parts of it
- Over multiple passes through the net (i.e. during training over many epochs):
  - Randomly dropping neurons “forces” each neuron to learn a non-trivial weight
  - The network can’t learn to rely on spurious correlations (i.e. meaningless patterns), because they randomly might not be present

Do we use dropout while testing? Why not?
Dropout: Implications for test time

• During testing, we stop dropping out and use all of the neurons again
Dropout: Implications for test time

• During testing, we stop dropping out and use all of the neurons again.
Dropout: Implications for test time

- During testing, we stop dropping out and use all of the neurons again.
- If a layer keeps a fraction $p$ of its neurons during training, then when we use all the neurons at test time, the next layer will get a bigger input than expected...
- **What do we do!?**
Dropout: Implications for test time

- **Solution 1:** Multiply the values of all neurons by $p$, so that the expected magnitude of the sum of neurons is the same.
Dropout: Implications for test time

- **Solution 1:** Multiply the values of all neurons by $p$, so that the expected magnitude of the sum of neurons is the same.
- **Solution 2:** At training time, divide the values of the kept neurons by $p$. 
Dropout - implementation

- Handy keras layer!

- `tf.keras.layers.Dropout(rate)`
  - Hyperparameter `rate` between [0, 1]: the rate at which the outputs of the previous layer are dropped
  - `Rate = 0.5`: drop half, keep half
  - `Rate = 0.25`: drop ¼, keep ¾

Any questions?
Dropout - why not?

- It’s invasive to the network – we’re “reaching inside” and directly modifying it
- Might be nice if we could get similar benefits without having to modify the network itself...
Overfitting: What can we do about it?

1. Early stopping
2. Reduce parameters
3. Data augmentation
4. Dropout
5. Regularization
Regularization - why?

This approach leaves the network architecture unchanged, and instead only modifies the **loss**.

- Adds an additional term to our existing loss function

Remember insight from before:

- Overfitting is correlated with the net relying **too heavily** on **too many different** correlations

Can we design a loss function that penalizes:

1. Heavy reliance on any correlation in the data?
2. Reliance on too many different correlations in the data?
Regularization - L1 vs L2

L2 regularization

• $\lambda \sum_{j=1}^{n} |W_j|^2$
  • Penalize sum of squared weights
  • **Effect**: keeps all weights small-ish, i.e. network can’t learn to rely too heavily on any single pattern in the data

L1 regularization

• $\lambda \sum_{j=1}^{n} |W_j|$
  • Penalize absolute value of weights
  • **Effect**: tends to produce sparse weights (i.e. many zero-valued weights) → prevents the network from relying on too many different patterns in the data

For both, this is a term added to the existing loss function.

$\lambda$ controls the strength of the penalty
Regularization - L1 vs L2

L1 Regularization

Cost = \sum_{i=0}^{N} (y_i - \sum_{j=0}^{M} x_{ij} W_j)^2 + \lambda \sum_{j=0}^{M} |W_j|

L2 Regularization

Cost = \sum_{i=0}^{N} (y_i - \sum_{j=0}^{M} x_{ij} W_j)^2 + \lambda \sum_{j=0}^{M} W_j^2

Loss function

Regularization Term
Regularization: implementation

• Implementing yourself
  • When calculating loss - Get list of model parameters, take L1/L2 norm, multiply by lambda – add to loss.

• If you only want to regularize the weights of certain layers:
  In `tf.keras`, regularization can be passed as an argument to the layer constructor:
  • `tf.keras.layers.Dense(16, kernel_regularizer=keras.regularizers.l1(lambda), activation='relu')`
Putting it all together

• Demo
TL;DR – Rule of Thumb for Overfitting:

“Make start with the small-ish architecture and your net bigger until it starts to overfit...

(use train/validation loss curves to monitor)

...then apply one of the techniques from this lecture”
Recap

Real-world data

Handling overfitting

- Messy and needs pre-processing
- Can lead to overfitting
- Early stopping may help but not the best solution

- Reduce parameters
- Data Augmentation
- Dropout
- Regularization

num_epochs

\[
\text{L1 Regularization} \\
\text{Cost} = \sum_{i=0}^{N} (y_i - \sum_{j=0}^{M} x_{ij} W_j)^2 + \lambda \sum_{j=0}^{M} |W_j|
\]

\[
\text{L2 Regularization} \\
\text{Cost} = \sum_{i=0}^{N} (y_i - \sum_{j=0}^{M} x_{ij} W_j)^2 + \lambda \sum_{j=0}^{M} W_j^2
\]

Loss function

Regularization Term